4. Aufgabenblatt zu Funktionale Programmierung vom 07.11.2012. Fällig: 14.11.2012 / 21.11.2012 (jeweils 15:00 Uhr)

Themen: *Funktionen auf algebraischen Datentypen*

Für dieses Aufgabenblatt sollen Sie die zur Lösung der unten angegebenen Aufgabenstellungen zu entwickelnden Haskell-Rechenvorschriften in einer Datei namens `Aufgabe4.lhs` im home-Verzeichnis Ihres Gruppenaccounts auf der Maschine g0 ablegen. Wie bei der Lösung zum dritten Aufgabenblatt sollen Sie auch dieses Mal ein "**literate Script**" schreiben. Versehen Sie wie auf den bisherigen Aufgabenblättern alle Funktionen, die Sie zur Lösung brauchen, mit ihren Typdeklarationen und kommentieren Sie Ihre Programme aussagekräftig. Benutzen Sie, wo sinnvoll, Hilfsfunktionen und Konstanten. Im einzelnen sollen Sie die im folgenden beschriebenen Problemstellungen bearbeiten.

Hinweis: Wenn Sie einzelne Rechenvorschriften aus früheren Lösungen wieder verwenden möchten, so kopieren Sie diese bitte in die neue Abgabedatei ein. Ein `import` schlägt für die Auswertung durch das Abgabeskript fehl, weil Ihre alte Lösung, aus der importiert wird, nicht mit abgesammelt wird. Deshalb: Kopieren statt importieren zur Wiederwendung!

Denken Sie bitte daran, dass Sie für die Lösung dieses Aufgabenblatts ein "literate" Haskell-Skript schreiben sollen!

Auf diesem Aufgabenblatt beschäftigen wir uns mit Darstellungen natürlicher und ganzer Zahlen und arithmetischen Operationen und Relationen darauf.

In Haskell können wir natürliche Zahlen durch folgenden algebraischen Datentyp realisieren:

```
data Nat = Z | S Nat deriving Show
```

Der Wert `Z` (kurz für "Zero") steht dabei für die Zahl 0, der Wert `S Z` (kurz für "Successor von Zero") für die Zahl 1, der Wert `S (S Z)` für die Zahl 2, der Wert `S (S (S Z))` für die Zahl 3, usw. Auf diese Weise besitzt jede natürliche Zahl eine eindeutige Darstellung als Wert des Datentyps `Nat`.

Schreiben Sie Haskell-Rechenvorschriften

1. `plusNat :: Nat -> Nat -> Nat`
2. `minusNat :: Nat -> Nat -> Nat`
3. `timesNat :: Nat -> Nat -> Nat`
4. `divNat :: Nat -> Nat -> Nat`
5. `modNat :: Nat -> Nat -> Nat`

zur Addition, Subtraktion, Multiplikation, ganzzahligen Division und zur Berechnung des Restes bei ganzzahliger Division. Dabei gilt für die Operation `minusNat`, dass das Resultat den Wert 0 hat, wenn das zweite Argument größer oder gleich dem ersten Argument ist. Das Resultat einer Division durch 0 bzw. einer Restbildung bezüglich 0 ist wie üblich nicht definiert. In beiden Fällen soll die Auswertung durch Aufruf von `error` mit dem Argument "Invalid argument" beendet werden.

Schreiben Sie weiters Haskell-Rechenvorschriften

6. `eqNat :: Nat -> Nat -> Bool`
7. `grNat :: Nat -> Nat -> Bool`
8. `leNat :: Nat -> Nat -> Bool`
9. `grEqNat :: Nat -> Nat -> Bool`
10. `leEqNat :: Nat -> Nat -> Bool`

die angewendet auf zwei Argumente überprüfen, ob die beiden Argumente gleich sind, das erste Argument echt größer bzw. echt kleiner als das zweite Argument ist, das erste Argument größer oder gleich bzw. kleiner oder gleich als das zweite Argument ist.

Folgende Beispiele zeigen das gewünschte Ein-/Ausgabeverhalten:

```
plusNat (S (S Z)) (S (S (S Z))) ->> S (S (S (S Z))))
minusNat (S (S Z)) (S (S (S Z))) ->> Z
timesNat (S (S Z)) (S (S (S Z))) ->> S (S (S (S (S Z)))))
divNat (S (S Z)) (S (S (S Z))) ->> Z
modNat (S (S Z)) (S (S (S Z))) ->> S (S Z)
eqNat (S (S Z)) (S (S (S Z))) ->> False
grNat (S (S Z)) (S (S (S Z))) ->> False
leNat (S (S Z)) (S (S (S Z))) ->> True
grEqNat (S (S Z)) (S (S (S Z))) ->> False
leEqNat (S (S Z)) (S (S (S Z))) ->> True
```

Unter Rückgriff auf den Datentyp `Nat` können wir ganze Zahlen als Paare natürlicher Zahlen darstellen. Wir führen dafür das Typsynonym `NatPair` ein:

```
type NatPair = (Nat,Nat)
```

Die durch ein Paar (m, n) aus `NatPair` dargestellte Zahl ergibt sich, in dem der Wert von n vom Wert von m abgezogen wird. So repräsentiert z.B. `((S (S Z)),(S (S (S (Z)))))` die Zahl -1. Ebenso repräsentiert `(Z,S Z)` die Zahl -1. Der Wert `(S Z,S Z)` repräsentiert die Zahl 0, ebenso wie der Wert `(Z,Z)`. Ganze Zahlen haben also als Werte des Datentyps `NatPair` keine eindeutige Darstellung.

Eine eindeutige Zahldarstellung in `NatPair` erhalten wir mit folgenden Zusatzvereinbarungen:

- Die Zahl 0 hat die Darstellung `(Z,Z)`.
- Echt positive Zahlen haben die Darstellung `(S (S (... Z)...),Z)`
- Echtnegative Zahlen haben die Darstellung `(Z,S (S (... Z)...))`

Wir bezeichnen diese eindeutige Darstellung einer ganzen Zahl durch einen Wert als ihre *kanonische* Darstellung in `NatPair`.

Schreiben Sie Haskell-Rechenvorschriften

11. `mkCan :: NatPair -> NatPair`
12. `plusNP :: NatPair -> NatPair -> NatPair`
13. `minusNP :: NatPairt -> NatPair -> NatPair`
14. `timesNP :: NatPair -> NatPair -> NatPair`
15. `divNP :: NatPair -> NatPair -> NatPair`
16. `modNP :: NatPair -> NatPair -> NatPair`

zur Überführung einer ganzen Zahl in seine eindeutige kanonische Darstellung, sowie zur Addition, Subtraktion, Multiplikation, ganzzahligen Division und zur Berechnung des Restes bei ganzzahliger Division, wobei das Resultat stets in kanonischer Form ausgegeben wird. Das Resultat einer Division durch 0 bzw. einer Restbildung bezüglich 0 ist wie üblich nicht definiert. In beiden Fällen soll die Auswertung durch Aufruf von `error` mit dem Argument "Invalid argument" beeendet werden.

Schreiben Sie weiters Haskell-Rechenvorschriften

17. `eqNP :: NatPair -> NatPair -> Bool`
18. `grNP :: NatPair -> NatPair -> Bool`
19. `leNP :: NatPair -> NatPair -> Bool`
20. `grEqNP :: NatPair -> NatPair -> Bool`
21. `leEqNP :: NatPair -> NatPair -> Bool`

die angewendet auf zwei Argumente überprüfen, ob die beiden Argumente gleich sind, das erste Argument echt größer bzw. echt kleiner als das zweite Argument ist, das erste Argument größer oder gleich bzw. kleiner oder gleich als das zweite Argument ist.

Beachten Sie: Alle Rechenvorschriften zur Behandlung ganzer Zahlen müssen sowohl mit Argumenten in kanonischer wie in nicht-kanonischer Darstellung umgehen können.

Folgende Beispiele zeigen das gewünschte Ein-/Ausgabeverhalten:

```
mkCan (S (S Z),S (S (S Z))) ->> (Z,S Z)
plusNP (S Z,S (S (S Z))) (S (S (S (S Z))),S Z) ->> (S Z,Z)
minusNP (S Z,S (S (S Z))) (S (S (S (S Z))),S Z) ->> (Z,S (S (S (S (S Z)))))
timesNP (S Z,S (S (S Z))) (S (S (S (S Z))),S Z) ->> (Z,S (S (S (S (S (S Z))))))
divNP (S Z,S (S (S Z))) (S (S (S (S Z))),S Z) ->> (Z,S Z)
modNP (S Z,S (S (S Z))) (S (S (S (S Z))),S Z) ->> (S Z,Z)
eqNP (S Z,S (S (S Z))) (S (S (S (S Z))),S Z) ->> False
grNP (S Z,S (S (S Z))) (S (S (S (S Z))),S Z) ->> False
leNP (S Z,S (S (S Z))) (S (S (S (S Z))),S Z) ->> True
grEqNP (S Z,S (S (S Z))) (S (S (S (S Z))),S Z) ->> False
leEqNP (S Z,S (S (S Z))) (S (S (S (S Z))),S Z) ->> True
```

Haskell Live

Am Freitag, den 09.11.2012, werden wir uns in *Haskell Live* mit Lösungsvorschlägen für Aufgabenblatt 1 und 2 beschäftigen, die (gerne auch) von Ihnen eingebracht werden können, sowie mit einigen der schon speziell für *Haskell Live* gestellten Aufgaben.